VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Interaktivní systém pro řešení příkladů výrokové logiky

Interactive System For Solving Propositional Logic Examples

2012 Ladislav Barabáš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ladislav Barabáš**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Interaktivní systém pro řešení příkladů výrokové logiky
Interactive System For Solving Propositional Logic Examples

Zásady pro vypracování:

Cílem bakalářské práce je vytvořit interaktivní systém pro řešení příkladů výrokové logiky pro tyto oblasti:
1. Důkazy ekvivalentními úpravami.
2. Důkazy pomocí rezoluční metody.

Aplikace by měla po vložení příkladu (formule či množiny formulí výrokové logiky) umožnit studentovi postupně provádět řešení úlohy a kontrolovat jeho postup. V případě, že student nebude vědet, jak pokračovat, nabídne mu aplikace nápovědu pro možnost dalšího postupu. Aplikace si bude ukládat správně vyřešené úlohy do své báze znalostí, aby je pak mohla nabízet jako vzorové příklady řešení.

Písemná část bakalářské práce bude obsahovat teorii a popis činnosti vytvořené aplikace:
1. Popis relevantních částí systému výrokové logiky.
2. Postup provádění transformací.
3. Popis rezoluční metody dokazování.
4. Popis algoritmu pro kontrolu řešení.
5. Popis návrhu implementace.
6. Popis ovládání aplikace.

Dokumentace k aplikaci musí být natolik podrobná, aby bylo možno na ni případně navázat při rozšiřování systému o další skupiny úloh.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jaromír Chocholatý**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Rád bych touto cestou poděkoval svému vedoucímu, Ing. Jaromíru Chocholatému, za jeho aktivní přístup, odborné vedení mé práce, cenné rady a čas, který mi věnoval.

Abstrakt

Tato práce pojednává o důkazových metodách výrokové logiky. Jedním z hlavních bodů je seznámení čtenáře se základy sémantiky a syntaxe výrokové logiky (dále jen VL), uvedení jej do dané problematiky a definování všech důkazových metod, které lze ve VL aplikovat. Cílem práce je vytvoření interaktivního systému, který pracuje s příklady výrokové logiky a k jejich řešení využívá důkazy ekvivalentními úpravami a důkazy pomocí rezoluční metody. Systém vede uživatele postupnými kroky ke správnému řešení daného příkladu. Všechny správně vyřešené příklady lze kdykoliv zpětně zobrazit, takže jsou k dispozici i vzorová řešení příkladů. Posledním bodem této práce je vytvoření podrobné uživatelské příručky, která slouží jako manuál k obsluze vytvořeného systému a také popisuje celý postup při jeho implementaci.

Klíčová slova: výroková logika, ekvivalentní úpravy, rezoluční metoda, formule, syntaktická analýza, syntaktický strom, JAVA

Abstract

This thesis discusses methods of proof of propositional logic. One key point is to acquaint readers with the basic of syntax and semantics of propositional logic (hereafter VL), putting it into this issue and define all methods of proof that can be applied in VL. The aim is to create an interactive system, that works with examples of propositional logic and their solutions are used by equivalent statements proof and proof by the resolution. The system guides the user in phases to the correct solution to the example. All examples can be properly resolved at any time re-appear, so there are also examples of sample solution. The last point of this work is to create detailed user guide, which serves as a manual for created system and also describes the procedure for its implementation.

Keywords: Propositional Logic, Equivalence Statements, Resolution, Formula, Syntactic Analysis, Syntactic Tree, JAVA

Seznam použitých zkratek a symbolů

VL	–	výroková logika
SA	–	syntaktická analýza
EBNF	–	Extended Backus-Naur Form
DNF	–	disjunktivní normální forma
KNF	–	konjunktivní normální forma
JAVA	–	programovací jazyk JAVA
IDE	–	Integrated Development Environment
XML	–	Extensible Markup Language
TXT	–	textový soubor

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Úvodní kapitola popisuje teorii jednotlivých částí systému VL. Čtenář je seznámen se základními pojmy, které blíže definují danou problematiku. Dále je zde popsána sémantika a syntaxe VL a také definovány důkazové metody, kterými lze ověřit splnitelnost formule. Celá tato kapitola má za úkol uvést teoretickou část VL, která se stala podkladem při implementaci jednotlivých částí systému.

Následující kapitola se zabývá popisem interaktivního systému, který byl vytvořen v rámci této bakalářské práce. Jednotlivé podkapitoly pojednávají o abecedě definované ve vytvořené aplikaci, popisují algoritmus syntaktické analýzy a také detailně vysvětlují všechny transformace, které jsou provedeny se vstupní formulí při převodu do výsledné formy. Dále je v textu popsán algoritmus, který provádí kontrolu vstupní formule při porovnávání s jednotlivými kroky řešení. Celá tato obsáhlá kapitola slouží jako detailní popis postupu při implementaci vytvořeného systému. Tento popis může posloužit jako podklad pro navázání při rozšiřování systému o další skupiny úloh.

Další kapitolou je popis návrhu implementace. Zde je blíže popsáno, jaké zdroje informací mi byly inspirací při vytváření systému a také jaké nástroje jsem v rámci této práce využil. Dále zde popisuji organizaci systému z hlediska zdrojového kódu.

Poslední kapitola je věnována uživatelskému rozhraní systému. Čtenáři je zde nabídnut popis funkčnosti jednotlivých komponent grafického rozhraní. Tato kapitola slouží především jako manuál pro orientaci v grafickém prostředí a blíže seznamuje uživatele s ovládáním systému.

2 Výroková logika

V této kapitole jsem vycházel ze studijních materiálů podle [1, 2].

Logika je věda, která se svou podstatou dotýká nás všech. Ať chceme či nechceme, logické myšlení používáme každý den a logikou se také každý den necháváme ovlivňovat. Logika ovšem ale není úzce spjata pouze s našimi životy, ale tato věda se dotýká všech oborů zkoumání. Zabývá se usuzováním, odvozování, dokazatelností a také vyvratitelností. V logice není důležitý obsah tvrzení, nýbrž forma sdělení.

Jedním ze základních odvozovacích systémů logiky je VL. Jedná se o logiku nultého řádu, ve které nemají výroky žádnou vnitřní stavbu a jediným jejich prvkem je pravdivostní hodnota. VL se zabývá analýzou vět do úrovně elementárních výroků. Zkoumá vytváření složených výroků z jednoduchých výroků a to pomocí logických spojek.

2.1 Výrok a formule

Základním prvkem VL je **výrok**. Výrok lze definovat jako oznamovací větu, u které má smysl zabývat se otázkou, zdali je pravdivá či nikoli (podle toho se označuje výrok jako pravdivý nebo nepravdivý). Výroky se dělí na atomické (jednoduché) a složené. Atomické výroky už dále nelze rozdělit na další výroky. Složené výroky se skládají z více výroků spojené logickými spojkami. Za výrok ovšem nelze považovat tázací větu či větu, u které nelze jednoznačně určit její pravdivostní hodnotu. Každý výrok je zároveň i **formule**. Jedná-li se o atomický výrok, lze jej označit jako atomickou formuli, jedná-li se o složený výrok, lze jej označit za složenou formuli.

2.2 Jazyk VL

2.2.1 Abeceda VL

Abeceda je tvořena skupinami symbolů, které reprezentují:

- výrokové symboly: p, q, r, s, ... (lze využít malá i velká písmena)
- logické spojky: negace, konjunkce, disjunkce, implikace, ekvivalence
- pomocné symboly: (,), [,], {, }

2.2.2 Gramatika VL

Gramatika VL udává, jak vytvářet formule podle následujících tvrzení:

- Každý výrokový symbol je formule. Takto vzniklé formule nazýváme atomické formule.
- Pokud jsou výrazy A, B formule, pak jsou také formule i výrazy: $\neg$A, (A $\wedge$ B), (A $\vee$ B), (A $\supset$ B), (A $\equiv$ B). Takto vzniklé formule nazýváme složené formule. Jednotlivé formule jsou v rámci složené formule označeny metasymboly (A, B, ...).

2.2.3 Sémantika VL

- Pravdivostní ohodnocení (**valuace**) výrokových proměnných lze chápat jako zobrazení v, které přiřazuje každé výrokové proměnné hodnotu z množiny pravdivostních hodnot $\{0, 1\}$. Pokud známe výrok označený symbolem p, můžeme jej označit jako pravdivý nebo nepravdivý. Podle toho položíme $v(\mathrm{p}) = 1$ nebo $v(\mathrm{p}) = 0$. Pokud ale pravdivostní hodnotu výroku p neznáme, pak můžeme položit $v(\mathrm{p}) = 1$ nebo $v(\mathrm{p}) = 0$, ovšem omezujeme se na situace, kdy je výrok p označený jako pravdivý nebo nepravdivý. Pak se lze zabývat, co plyne z pravdivosti či nepravdivosti výroku p.

- Pravdivostní funkce formule VL lze chápat jako funkci w, která jednoznačně přiřazuje pravdivostní hodnotu celé výrokové formule ke každému pravdivostnímu ohodnocení v výrokových proměnných.

  - Pokud se pravdivostní hodnota výrokové formule A rovná pravdivostnímu ohodnocení výrokové proměnné p, pak A = p, tj. $w(\mathrm{A})_v = v(\mathrm{p})$.
  - Známe-li pravdivostní funkce formulí A, B, lze jednoduše zjistit pravdivostní funkce formulí $\neg$A , (A $\wedge$ B), (A $\vee$ B), (A $\supset$ B), (A $\equiv$ B) podle následujících pravidel:

A	B	$\neg$A	A $\wedge$ B	A $\vee$ B	A $\supset$ B	A $\equiv$ B
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

2.3 Převod z přirozeného jazyka do jazyka VL

Přirozený jazyk se převádí do jazyka VL na základě analýzy struktury vět do úrovně elementárních výroků. Při tomto převodu se z běžné věty stává formule VL, popř. formule predikátové logiky. Tento proces je označován jako **formalizace**. Převedená věta již nevyjadřuje původní obsah, ale pouze strukturu věty, která se nazývá logická forma věty. Pro označení elementárních výroků v dané větě se používají různé výrokové symboly (viz. kapitola 2.2.1). Jednotlivé elementární výroky jsou v rámci věty na sobě nezávislé a jsou určeny pouze svou pravdivostní hodnotou. Spojky přirozeného jazyka jsou nahrazeny logickými spojkami. Analogie mezi těmito spojkami je následující:

2.3.1 Unární spojky

2.3.1.1 Negace

Negace se používá, pokud se v přirozeném jazyce vyskytuje zápor nebo slovní spojení *„není pravda, že“*. Ve VL se negace značí symbolem $\neg$.

$$\textit{Není pravda, že jsem byl ve škole.} \implies \neg p$$

2.3.2 Binární spojky

2.3.2.1 Konjunkce

Konjunkce je analogií spojky *„a“* přirozeného jazyka. Tato analogie ovšem není podmínkou. V určitých případech se o konjunkci nejedná. Konjunkce se označuje symbolem $\wedge$.

$$\textit{Dnes jsem nešel do školy a zůstal jsem doma.} \implies p \wedge q$$
$$\textit{Petr a Pavel byli ve škole.} \quad \text{nejedná se o konjunkci}$$

2.3.2.2 Disjunkce

Logická spojka disjunkce odpovídá spojce *„nebo“*. Pozor je třeba dávat na situace, kdy má spojka *„nebo“* vylučovací charakter ve smyslu *„buď“*, *„anebo“*, pak tato spojka představuje nonekvivalenci při převodu do VL. Symbol charakterizující disjunkci je $\vee$.

$$\textit{Musím spěchat do školy nebo přijdu pozdě.} \implies p \vee q$$
$$\textit{Půjdu do školy, nebo zůstanu doma.} \implies \neg(p \equiv q) \quad \text{vylučuje se}$$

2.3.2.3 Implikace

Tato spojka se označuje symbolem $\supset$ a vyjadřuje výrokovou vazbu *„jestliže...,pak...“*, *„když...,tak...“*, *„je-li...,pak...“*. Implikace je jediná binární spojka, která není komutativní. To znamená, že členy implikace nelze svévolně převracet tak, abychom neztratili jejich význam. Můžeme si tedy dovolit pojmenovat její členy – první člen se nazývá **antecedent** (příčina), druhý je **konsekvent** (důsledek). Mezi těmito členy se nepředpokládá žádná obsahová ani časová souvislost, takže lze spojit implikací dva výroky, aniž by spolu musely nějakým způsobem souviset.

$$\textit{Když půjdu dnes do školy, musím si udělat svačinu.} \implies p \supset q$$
$$\textit{Je-li venku hezky, jmenuji se Pavel.} \implies p \supset q \quad \text{bez bližší souvislosti}$$

2.3.2.4 Ekvivalence

Poslední binární spojka ekvivalence vyjadřuje výrokovou vazbu *„právě tehdy, když“*, *„tehdy a jen tehdy, když“*. Ekvivalence se vyjadřuje symbolem $\equiv$. V určitých situacích je obtížné odlišit implikaci od ekvivalence. Například vazba *„tehdy, když“* už není vyjádřením ekvivalence nýbrž implikace. Rozdíl mezi těmito spojkami je velmi důležitý pochopit,

jelikož převážná většina tvrzení v matematice jsou buď ve tvaru implikace, nebo ekvivalence.

Dostanu odměnu právě tehdy, když budu hodný. $\implies p \equiv q$
Dostanu odměnu tehdy, když budu hodný. $\implies p \supset q$

2.4 Důkazové metody ve VL

Důkazové metody slouží k zjištění pravdivostní hodnoty celé formule. Lze tedy jimi zjisti, zdali je daná formule splnitelná, nesplnitelná či tautologie. Mezi tyto metody patří **tabulková metoda**, **důkaz sporem** a **rezoluční metoda** (viz. kapitola 2.4.3). První dvě zmíněné metody je vhodné aplikovat na formule, které mají menší počet různých výrokových proměnných. Rezoluční metodu je efektivní použít tam, kde se vyskytuje větší množství různých výrokových symbolů.

2.4.1 Tabulková metoda

Jak již z názvu vyplývá, tato metoda je založena na vytvoření tabulky, která obsahuje všechny možné pravdivostní ohodnocení výrokových proměnných ve formuli. Lze ji aplikovat na jakoukoli formuli, avšak vhodné je použít ji tam, kde se vyskytuje malý počet různých výrokových proměnných ve formuli, protože s přibývajícími různými výrokovými symboly roste také počet řádků tabulky a to exponenciálně. Pokud tedy máme n různých výrokových proměnný ve formuli, pak počet pravdivostních hodnot je 2^n a tedy i tabulka bude mít 2^n řádků. Takže například na formuli s deseti různými atomy by bylo nesmyslné aplikovat tuto metodu, jelikož bychom museli vytvořit tabulku s 1024 řádky a v každém vypočítat hodnotu formule.

2.4.2 Metoda protipříkladu

Metoda protipříkladu, někdy také označována jako důkaz sporem, je pokročilejší metoda při ověřování tautologie ve tvaru implikace a při ověřování logického vyplývání. Protipříkladem k formuli A budeme rozumět takovou valuaci formule A, která je rovna nule, tedy $v(A) = 0$. Tuto metodu je efektivní aplikovat na formule, které mají větší počet různých výrokových proměnných.

2.4.3 Rezoluční metoda

Pomocí této metody lze ověřit platnost úsudku nebo také rozhodnout, zdali je zadaná formule (resp. množina formulí) splnitelná nebo nesplnitelná. Rezoluční metodu lze aplikovat na formuli v KNF.

2.4.3.1 Pravidla rezoluční metody

- Pokud formule A je tautologie, pak je formule $\neg A$ kontradikcí a naopak.

- Nechť l je literál. Pro odvození $(A \vee B)$ z formule $(A \vee l) \wedge (B \vee \neg l)$ se uplatňuje rezoluční pravidlo odvozování, zapisujeme takto:

$$\frac{(A \vee l) \wedge (B \vee \neg l)}{(A \vee B)}$$

  Toto pravidlo nevede k ekvivalentní formuli předcházející formule, ale zachovává její splnitelnost. Odvozená formule se nazývá **rezolventa**.

2.4.3.2 Klauzulární forma formule

V rezoluční metodě se jednotlivé konjunkty nazývají **klauzule**. Proto se označuje daná formule jako **klauzulární forma**. Klauzulární forma je tedy jiný název pro formuli v KNF. Kupříkladu formule $(p \supset q) \wedge (q \supset (p \vee q)) \wedge (r \vee p)$ má 3 klauzule.

2.4.3.3 Přímý důkaz

Máme-li úsudek s předpoklady $P_1, \ldots P_n \models Z$, pak na předpoklady tohoto úsudku uplatňujeme rezoluční pravidlo, tedy odvozujeme další formule $(P_n + 1, \ldots, P_m)$ jakožto jejich důsledky. Aby byl úsudek platný, musí být poslední odvozená formule P_m shodná se závěrem Z.

Příklad 2.1
Nyní ověříme platnost úsudku $p \vee q, p \supset r \models \neg r \supset q$. Každou klauzuli si napíšeme pod sebe a aplikujeme rezoluční pravidlo.

1.	$p \vee q$		
2.	$p \supset r$	$\Longleftrightarrow$	$\neg p \vee r$
3.	$\neg r \supset q$	$\Longleftrightarrow$	$r \vee q$
4.	$q \vee r$	(1, 2)	■

Poslední odvozená formule je shodná se závěrem úsudku. Úsudek můžeme označit za platný. ■

2.4.3.4 Nepřímý důkaz

Mějme úsudek s předpoklady $P_1, \ldots, P_n$, ze kterých plyne závěr Z, tj. $P_1, \ldots, P_n \models Z$. Pak se nepřímý důkaz provádí tak, že nejdříve znegujeme závěr Z a dokazujeme, že množina $M = \{P_1 \ldots, P_n, \neg Z\}$ je sporná.

Příklad 2.2
Nyní ověříme platnost úsudku $p \vee q, p \supset r \models \neg r \supset q$. V tomto případě si pod sebe již napíšeme pouze předpoklady úsudku a poté znegujeme závěr a připojíme jej k těmto předpokladům.

1.	$p \vee q$		
2.	$p \supset r$	$\Longleftrightarrow$	$\neg p \vee r$
3.	$\neg r$		negovaný závěr
4.	$\neg q$		negovaný závěr
5.	$q \vee r$	(1, 2)	
6.	r	(4, 5)	
7.	$\square$	(3, 6)	

Jak je z příkladu patrné, v posledním bodě došlo ke sporu, což znamená, že je negovaný závěr ve sporu s předpoklady, a proto je úsudek platný. ■

2.5 Normální formy formulí

Ke každé formuli A výrokové logiky je přiřazena právě jedna pravdivostní funkce *w*. K této funkci ovšem existuje více formulí, které jsou její reprezentací. Tyto formule jsou proto navzájem ekvivalentní. Podle definice jsou formule navzájem ekvivalentní, pokud mají přesně stejné modely, tj. mají stejnou pravdivostní hodnotu. Abychom předešli nejednoznačnosti těchto formulí, zavádíme tzv. **normální tvar formule**. Množina ekvivalentních formulí je pak reprezentována tímto normálním tvarem formule.

Následující definice byly převzaty ze studijních materiálů [1].

Definice 2.1

- ***Literál*** *je výroková proměnná nebo jeho negace.*
- ***Elementární konjunkce*** *(EK) je konjunkce literálů.*
- ***Elementární disjunkce*** *(ED) je disjunkce literálů.*
- ***Úplná elementární konjunkce*** *(UEK) dané množiny výrokových symbolů je elementární konjunkce, ve které se každý symbol z dané množiny vyskytuje právě jednou.*
- ***Úplná elementární disjunkce*** *(UED) dané množiny výrokových symbolů je elementární disjunkce, ve které se každý symbol z dané množiny vyskytuje právě jednou.*
- ***Disjunktivní normální forma*** *(DNF) dané formule je formule ekvivalentní s danou formulí a mající tvar disjunkce elementárních konjunkcí.*
- ***Konjunktivní normální forma*** *(KNF) dané formule je formule ekvivalentní s danou formulí a mající tvar konjunkce elementárních disjunkcí.*
- ***Úplná disjunktivní normální forma*** *(UDNF) dané formule je formule ekvivalentní s danou formulí a mající tvar disjunkce úplných elementárních konjunkcí.*
- ***Úplná konjunktivní normální forma*** *(UKNF) dané formule je formule ekvivalentní s danou formulí a mající tvar konjunkce úplných elementárních disjunkcí.*

3 Aplikace Logik

Aplikace Logik slouží k řešení vybraných příkladů z výrokové logiky a lze ji využít jako pomůcku při studiu matematické logiky. Tato aplikace provádí důkazy ekvivalentními úpravami a důkazy rezoluční metodou (přímý důkaz, nepřímý důkaz). Po zadání formule či množiny formulí do systému lze s aplikací pracovat ve dvou módech: základní a interaktivní. Interaktivní mód poskytuje přímou vazbu s uživatelem, jelikož jej aplikace vede postupnými kroky ke správnému řešení daného příkladu. Všechny správně řešené příklady se ukládají do databáze aplikace, která slouží jako sbírka příkladů. Později lze skrze aplikaci vybírat z databáze jednotlivé příklady, které mohou posloužit jako vzorové řešení.

3.1 Abeceda aplikace

Při sestavování formule lze v aplikaci použít různé symboly, které vycházejí z obecně používané abecedy VL (viz. kapitola 2.2.1). Pro výrokové symboly jsou přirozeně vyhrazena písmena (malá i velká). Logické spojky VL jsou pak nahrazeny alternativními znaky z klávesnice, které se poměrně často používají v praxi. Z pomocných symbolů jsou v aplikaci povoleny kulaté závorky.

Abeceda VL a její alternativní symboly v aplikaci:

Skupina	Symbol ve VL	Symbol v aplikaci	Poznámka
výrokové symboly	$A \dots Z, a \dots z$	$A \dots Z, a \dots z$	
logické symboly	$\neg$	$\sim$	negace
logické symboly	$\wedge$	$\&$	konjunkce
logické symboly	$\vee$	$\mid$	disjunkce
logické symboly	$\supset$	$>$	implikace
logické symboly	$\equiv$	$=$	ekvivalence
pomocné symboly	$(,)$	$(,)$	

Tabulka 1: Alternativní symboly

3.2 Syntaktická analýza

V této kapitole jsem vycházel ze studijních materiálů podle [3].

Syntaktická analýza (SA) je první důležitý krok při provádění jakýchkoli operací s formulí. Pomocí ní lze jednoduše zjistit, jestli má výraz na vstupu syntakticky správný tvar. Algoritmus, který je reprezentací SA, se nazývá **syntaktický analyzátor**. Zadá-li uživatel na vstupu určitou formuli, pak pomocí analyzátoru proběhne kontrola, jestli je ve správném tvaru. Pokud je tedy formule korektně zapsána, analyzátor vytvoří tzv. **syntaktický strom** (též derivační strom), který reprezentuje strukturu vstupní formule. S tímto

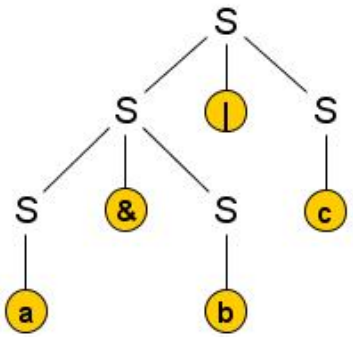
stromem pak lze provádět další transformace. Pokud ovšem formule není ve správném tvaru, analyzátor oznámí chybu a aplikace následně vyzve uživatele, aby změnil zadání.

3.2.1 Metoda rekurzivního sestupu

Pro vytvoření syntaktického stromu je v aplikaci použita **metoda rekurzivního sestupu**. Tato metoda vychází z jedné ze základních metod SA – metody shora dolů. Syntaktický strom je konstruován „shora dolů“ tedy od kořene k listům pomocí levé derivace. V následujícím příkladu je zobrazen postup provádění derivace (pouze pro jednoduchý model gramatiky).

Příklad 3.1
$G = (\{S\}, \{a, b, c, \&, |\}, P, S)$
$P : S \rightarrow S\&S \mid S|S \mid a \mid b \mid c$
Chceme-li z této gramatiky utvořit například formuli a& b | c, provedeme následující derivaci:
$S \rightarrow S|S \rightarrow S\&S|S \rightarrow a\&S|S \rightarrow a\&b|S \rightarrow a\&b|c$

Jak je z příkladu patrné, v každém kroku odvození se přepíše jeden neterminální symbol, který byl nejvíce vlevo. Tento postup se nazývá levá derivace. Výsledný strom pak graficky vypadá takto:



Obrázek 1: Syntaktický strom

■

Podstatou metody rekurzivního sestupu je vytvoření několika metod programovacího jazyka podle **rozvinuté Backusovy–Naurovy formy** (EBNF). Tato forma se užívá zvláště v informatice k vyjádření bezkontextové gramatiky používané pro formální popis programovacích jazyků.

Syntaxe EBNF je velmi podobná bezkontextové gramatice. Neterminály se zapisují do úhlových závorek a přepisují se skrze symbol ::= na řetězce terminálů a neterminálů. Složené závorky za výrazem značí, že se výraz může vyskytovat v libovolném počtu. Gramatika v EBNF pro názvy implementovaných metod v aplikaci vypadá následovně:

<*expEkviv*> ::= <*expImpl*> {= <*expImpl*>}
<*expImpl*> ::= <*expDisj*> {> <*expDisj*>}
<*expDisj*> ::= <*expConj*> {| <*expConj*>}
<*expConj*> ::= <*expOper*> {& <*expOper*>}
<*expOper*> ::= <*expOper*> | (<*expEkviv*>) | 0 | 1 | | A | ... | Z | a | ... | z

Každá takto implementovaná metoda je přiřazena právě jednomu neterminálu EBNF. Jinými slovy, pro každou logickou spojku (kromě negace) je vytvořena právě jedna metoda, v této metodě je volána metoda *getChar* (pro načtení následujícího symbolu) a každá metoda obsahuje rekurzivní volání metody, představující logickou spojku s nižší prioritou. Výjimkou je metoda *expConj*, která reprezentuje logickou spojku konjunkce a jelikož další spojka s nižší prioritou už není, volá metodu *expOper*, ve které proběhne SA operandu. Následující fragment kódu popisuje metodu *expConj*.

```
public void expConj() {
    if (error.equals(Errors.OK)) {
        expOper();
        while (ch == Chars.AND) {
            getChar();
            expOper();
            if (error.equals(Errors.OK)) {
                setLocCont(formula.putOper(localContainer, Chars.AND,
                    Chars.priorAND));
            }
        }
    }
}
```

Výpis 1: Metoda expConj

3.2.2 Syntaktický analyzátor

Syntaktický analyzátor je v aplikaci Logik zastoupen třídou **Parser**. Tato třída je zároveň jakýmsi jádrem celé aplikace, jelikož jsou v ní zapouzdřeny všechny operace potřebné k vykonání důkazu pro danou oblast VL. Konstruktor této třídy obsahuje jeden parametr a tím je pole znaků. Toto pole představuje vstupní formuli, která je převedena na jednotlivé znaky a ty jsou uloženy do pole.

Metoda *parse* je hlavní metoda třídy Parser. Zapouzdřuje všechny operace SA a také všechny další operace, které je nutné vykonat při výpočtech. O větvení bloků operací se stará příkaz switch, pomocí kterého se jednoduše rozhoduje, zdali se má provést důkaz ekvivalentními úpravami či rezoluční metodou.

Nezbytnou součástí syntaktického analyzátoru je lexikální analyzátor, který načítá jednotlivě symboly z pole znaků (vstupní formule). Tento analyzátor je zastoupen bezparametrickou metodou *getChar*, která ukládá právě čtený znak do proměnné *ch* a v případě načtení nepřípustného znaku ukládá do proměnné *error* chybové hlášení (*neplatný znak*). Proměnná *position* určuje pozici aktuálně čteného symbolu.

```
public void getChar() {
    if (( position < charArray.length)) {
        ch = charArray[position];
        if (!(( ch >= 'a') && (ch <= 'z') || (ch >= 'A')
            && (ch <= 'Z') || (ch == '0') || (ch == '1') ||
            (ch == Chars.AND) || (ch == Chars.OR) || (ch == Chars.NEG) ||
                (ch == Chars.IMPL) || (ch == Chars.EKV) || (ch == Chars.L_BRA) ||
                (ch == Chars.R_BRA) && (error.equals(Errors.OK)))) {
                    error = Errors.ILL_CHAR;
        }
        position++;
    } else {
        ch = '\n';
    }
}
```

Výpis 2: Lexikální analyzátor

Syntaktický analyzátor dále definuje prioritu vložených znaků. Nejnižší priorita je přiřazena výrokové proměnné, nejvyšší prioritu má spojka ekvivalence. Při vytváření formule je velmi důležité dbát na správné uzávorkování podvýrazů. Jednotlivé priority znaků jsou definovány v následujícím pořadí:

výroková proměnná, konjunkce, disjunkce, implikace, ekvivalence

3.2.3 Průběh SA v aplikaci

Máme-li na vstupu formuli p& q | r, SA v aplikaci proběhne podle následujících kroků:

- čtení znaku **p**
- uložení znaku **p** do kontejneru (slouží pro konstrukci syntaktického stromu)
- čtení znaku **q**
- uložení znaku **q** do kontejneru
- výběr z kontejneru posledních dvou výrazů (p, q) a spojení symbolem **&**
- čtení znaku **r**
- výběr z kontejneru posledních dvou výrazů (p& q, r) a spojení symbolem **|**

3.3 Stromová reprezentace logických výrazů

Vkládá-li uživatel do systému formuli, pak je tato formule zapsána v tzv. infixním tvaru čili tvaru přirozené notace. Tento tvar notace je lépe čitelný pro uživatele oproti tvaru postfixové notace, kde operátor následuje své operandy. Ovšem z programátorského hlediska by nebylo příliš vhodné zpracovávat zadanou formuli v infixním tvaru,

a proto syntaktický analyzátor převádí danou formuli na syntaktický strom, který lze efektivněji zpracovávat.

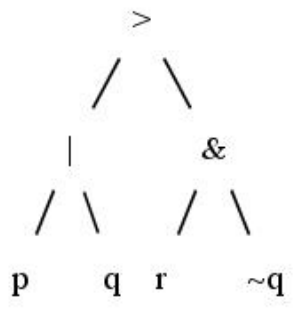
Syntaktický strom je reprezentací orientovaného grafu s ohodnocenými uzly a nejvyšším prvkem stromu – kořenem. Každý uzel může mít své potomky, takové uzly označujeme jako vnitřní uzly stromu. Uzly, které nemají potomky, se nazývají listy stromu.

Ohodnocení uzlů stromu:

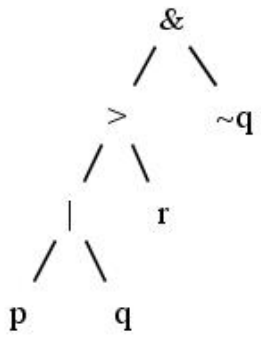
- vnitřní uzly stromu jsou operátory (ekvivalence, implikace, disjunkce, konjunkce)
- listy stromu jsou operandy (výrokové proměnné)

Příklad 3.2
Mějme strom reprezentující strukturu formule (p | q)>r&∼q.

Kořenem stromu je určena spojka implikace, jelikož má nejvyšší prioritu. Potomky tohoto kořene jsou podstromy | (disjunkce proměnných) a & (konjunkce proměnných). Do stromu není nutno vkládat pomocné symboly v podobě závorek, jelikož priorita operandů a operátorů je pevně dána v analyzátoru programu. Nicméně kulaté závorky jsou také důležitou pomůckou při sestavování formule, protože pomocí nich lze ovlivnit konstrukci stromu nezávislou na prioritě operátorů. Tak například, vezměme si znova formuli (p | q)>r&∼q a upravme ji takto ((p | q)>r)&∼q. Pak už zajisté nebude kořenem stromu spojka implikace, ale bude to spojka konjunkce, mající potomky > (implikace složené formule) a výrokový symbol ∼q. Nyní obě formule porovnejme graficky:



Obrázek 2: Strom formule (p | q)>r&∼q



Obrázek 3: Strom formule ((p | q)>r)&∼q

■

3.3.1 Konstrukce stromu

Aby mohl program vytvořit syntaktický strom, potřebuje mít k dispozici datový typ, do kterého lze ukládat jednotlivé uzly stromu a následně je mezi sebou propojovat logickými spojkami. V programu jsou pro tyto účely vytvořené dva objekty – **Node** a **Container**.

První zmíněný objekt reprezentuje uzel stromu, jehož atributy jsou: *character* (výrokový symbol), *priority* (priorita symbolu), *negation* (negace), *left* (levý uzel) a *right* (pravý uzel). Levý a pravý uzel odkazují na potomky uzlu. V případě listu stromu žádné potomky definovány nejsou, a proto jsou tyto atributy nastaveny na hodnotu null.

Objekt Container zaobaluje uzly do jednoho celku a vytváří tak stromovou strukturu. Tento objekt má dva atributy: *node* (typu Node) a *next* (typu Container). Při sestavování syntaktického stromu se vždy oba atributy vyberou, spojí logickou spojkou a výsledný uzel uloží zpět do kontejneru (předchozí atributy jsou vymazány).

3.3.2 Průběh konstrukce stromu v aplikaci

Mějme jednoduchou formuli p | q. Syntaktický analyzátor nejdříve uloží výrokové proměnné p a q do kontejneru pomocí funkce *put* třídy **Formula**. Ukládání probíhá tak, že se v kontejneru vytvoří nový uzel, tomu jsou nastaveny atributy předané v parametru funkce *put* a atributu *next* je přiřazen kontejner předaný taktéž v parametru funkce *put*. Výsledný kontejner po vložení proměnných p a q bude vypadat takto:

Container(node(q), next(node(p), next(null,null)))

Dále pak analyzátor pomocí funkce *putOper* vybere z kontejneru oba atributy a spojí je logickou spojkou | předanou v parametru této funkce. Vznikne nový uzel se symbolem logického operátoru |. Tento uzel má 2 potomky – listy p a q. Nakonec se uzel přiřadí atributu *node* objektu *Container*.

3.4 Důkazy ekvivalentními úpravami

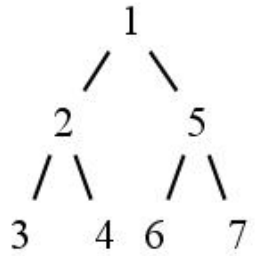
Když už je syntaktický strom vytvořen, lze s ním provádět různé transformace od jednodušších, jako je odstranění negace směrem na proměnné, až po ty složitější, jako je minimalizace formule. V této kapitole si popíšeme všechny druhy transformací, které proběhnou v aplikaci při provádění ekvivalentních úprav logických výrazů.

Při provádění transformací je nutné implementovat algoritmus, který zajistí průchod syntaktickým stromem v daném pořadí. Pokud chceme procházet syntaktický strom, máme dvě možnosti, jakou skupinu metod zvolit. Průchod do šířky a častěji používaný průchod do hloubky. Průchod do hloubky je definován rekurzivně a uzly stromů jsou zpracovávány v pořadí, které je dané jejich hloubkou, zatímco průchod do šířky je nerekurzivní a uzly stromů jsou navštěvovány podle jejich úrovní, ve kterých se vyskytují (kořen má úroveň 0). V každé úrovni jsou uzly zpracovány zleva doprava. Pro implementaci bylo výhodné využít skupinu průchodu do hloubky, která definuje tyto metody: **Pre-order**, **Post-order**, **In-order**.

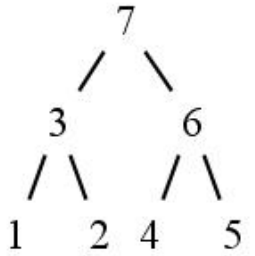
Metody průchodů do hloubky:

- **Pre-order** = vyhodnocování shora - nejdříve se zpracovává kořen a poté podstromy (potomci)

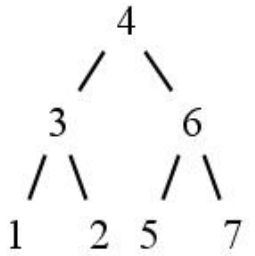
- **Post-order** = vyhodnocování zdola – nejdříve se zpracovávají podstromy a poté až kořen stromu
- **In-order** = systematické vyhodnocování – nejdříve se zpracovává levý podstrom, poté kořen a nakonec pravý podstrom



Obrázek 4: Pre-order



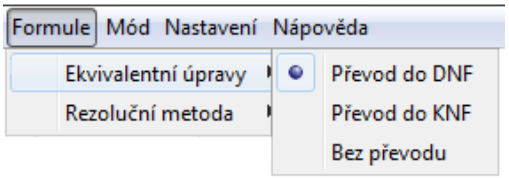
Obrázek 5: Post-order



Obrázek 6: In-order

Operace, které modifikují logický výraz při provádění ekvivalentních úprav, probíhají v aplikaci v tomto pořadí: vyhodnocení logických konstant, odstranění negace směrem na proměnné a eliminace logických spojek, převod do disjunktivní/konjunktivní normální formy, třídění a redukce formule, minimalizace formule, převod syntaktického stromu do infixní formy.

Aplikace Logik nabízí možnost vynechání provádění operací: převod do disjunktivní/konjunktivní normální formy a minimalizace formule. Důvodem je možnost provádění pouze elementárních úprav na úrovni eliminace logických spojek. V interaktivním módu aplikace (viz. kapitola 3.6) by totiž mohlo dojít ke zdlouhavému výpočtu vložené formule vlivem provádění minimalizace, což by mohlo být matoucí pro uživatele. Tyto operace transformace formule lze povolit/zakázat v hlavním menu. Výběr mezi převodem do normální formy (disjunktivní, konjunktivní) je realizován pomocí přepínačů, takže lze zvolit vždy jen jednu možnost. Výběrem „**Bez převodu**“ se zakáže převádění do určité normální formy a zároveň i minimalizace formule.



Obrázek 7: Menu - Ekvivalentní úpravy

3.4.1 Odstranění negace směrem na proměnné, eliminace logických spojek

Ve VL bývá pravidlem, že se při řešení ekvivalentních úprav snažíme „vytlačit“ negace směrem k výrokovým proměnným. Pro odstranění negace směrem na proměnné se využívají tyto zákony:

$$\neg(p \wedge q) \equiv (\neg p \vee \neg q) \qquad \text{De Morganovy zákony}$$
$$\neg(p \vee q) \equiv (\neg p \wedge \neg q) \qquad \text{De Morganovy zákony}$$

Dále se ve VL snažíme zjednodušit výraz a rozložit jej na co možno nejjednodušší tvar. Proto dochází k eliminaci složitějších logických spojek, jako jsou ekvivalence či implikace, které se převádí na konjunkce a disjunkce proměnných. Přitom mezi logickými spojkami jsou následující ekvivalence:

$$(p \equiv q) \equiv (p \supset q) \wedge (q \supset p) \qquad \text{eliminace ekvivalence}$$
$$(p \equiv q) \equiv (p \wedge q) \vee (\neg q \wedge \neg p) \qquad \text{eliminace ekvivalence}$$
$$(p \supset q) \equiv (\neg p \vee q) \qquad \text{eliminace implikace}$$
$$\neg(p \supset q) \equiv (p \wedge \neg q) \qquad \text{negace implikace}$$

Všechny tyto ekvivalence jsou implementovány do aplikace, která pracuje se syntaktickým stromem a algoritmus, který rekurzivně prochází strom, je zastoupen metodou *preOrder* třídy **ImpDisCon**. Průchod stromu je typu Pre-order čili se postupuje od kořene shora dolů. Pokud algoritmus narazí na negovaný vnitřní uzel, změní jeho symbol (dle pravidel), odstraní negaci a zároveň zneguje potomky uzlu. Převody mezi logickými spojkami jsou principiálně podobné. Narazí-li algoritmus na vnitřní uzel, reprezentující logickou spojku, kterou lze eliminovat, změní jeho symbol a dle ekvivalentních úprav také změní potomky uzlu.

Metoda *preOrder* obsahuje 3 vedlejší metody. Tyto metody jsou: *implication*, *disjunction*, *conjuction*. Z názvu metod vyplývá, že každá je určena pro jinou logickou spojku. Metoda *implication* provádí eliminaci ekvivalence na implikaci, metoda *disjunction* provádí eliminaci implikace na disjunkci a také odstraňuje negaci konjunkce směrem na proměnné a metoda *conjuction* odstraňuje negaci disjunkce směrem na proměnné.

3.4.2 Vyhodnocení logických konstant

Formule vkládaná do systému aplikace nemusí obsahovat pouze znaky výrokových proměnných a symboly logických spojek. Pokud známe ohodnocení výrokových proměnných ve formuli, je výhodnější místo znaků přímo vkládat hodnotu těchto proměnných. Tato ohodnocení jsou typicky označována jako pravda (1) a nepravda (0). Známe-li hodnoty některých proměnných, lze výraz vyhodnotit podle následujících pravidel:

negace	konjunkce	disjunkce	implikace	ekvivalence
$\neg 0 \equiv 1$	$p \wedge 0 \equiv 0$	$p \vee 0 \equiv p$	$p \supset 0 \equiv \neg p$	$p \equiv 0 \equiv \neg p$
$\neg 1 \equiv 0$	$p \wedge 1 \equiv p$	$p \vee 1 \equiv 1$	$p \supset 1 \equiv 1$	$p \equiv 1 \equiv p$
			$0 \supset p \equiv 1$	
			$1 \supset p \equiv p$	

Tabulka 2: Pravidla pro vyhodnocení logických konstant

Všechna tato pravidla jsou zavedena ve funkci *evalConst* třídy **Constant**. V této funkci se pracuje se dvěma pomocnými uzly – *nodeOperand* a *nodeConst*. Uzlu *nodeOperand* je přiřazena výroková proměnná (operand) a uzlu *nodeConst* je přiřazena logická konstanta dle implementovaných podmínek. Druhou část funkce tvoří rozpoznávání operátoru (pomocí větvení příkazem switch) a podle toho se zpracovává uzel předaný v parametru této metody. Důležitou roli také hraje lokální proměnná *leftImpl* typu boolean, která je využita při vyhodnocování implikace. Implikace je totiž jediná nekomutativní spojka, a proto je nutné rozlišit, v jakém pořadí se vyskytuje proměnná a konstanta. Výsledky implikace se v různém pořadí vložených symbolů různí. Proměnné *leftImpl* je přiřazena hodnota true, jestliže se logická konstanta nachází vlevo od logické spojky. Pokud je tomu naopak, je proměnné přiřazena hodnota false.

Procházení syntaktického stromu při vyhodnocování logických konstant je typu Post-order, čili od listů směrem ke kořeni stromu. Reprezentací tohoto průchodu je metoda *postOrder* třídy Constant. Při procházení syntaktického stromu se pracuje s lokálním stromem, který slouží jako jakýsi buffer, do kterého jsou ukládány změny provedené funkcí *evalConst*. Pokud buffer není prázdný, přiřadí se jeho obsah právě čtenému uzlu ze vstupního syntaktického stromu a následně se buffer opět vyprázdní. Takto dochází k přepisování syntaktického stromu při vyhodnocování logických konstant.

3.4.3 Převod syntaktického stromu do infixní formy

Výsledný syntaktický strom se všemi provedenými ekvivalentními úpravami je třeba umět zpětně vypsat v textové podobě, aby mohl být prezentován v aplikaci. K tomuto účelu je vytvořena funkce *convertInfix* třídy **Infix**. Tato funkce řeší výpis syntaktického stromu do infixní formy. K výpisu využívá průchod typu In-order, takže je konstruována tak, že se nejdříve prochází levý podstrom (rekurzivním voláním sama sebe) pro logickou spojku, pak se vždy ukládá znak atomu a nakonec se prochází pravý podstrom pro logickou spojku. Všechny přečtené znaky se ukládají do lokální proměnné *infixString* pomocí metod *insertString* a *insertChar*. Tato proměnná je návratovým typem funkce *convertInfix*.

Kromě ukládání znaků, které jsou součástí syntaktického stromu, řeší funkce *convertInfix* i ukládání symbolu negace a samozřejmě kulatých závorek, které jsou nezbytné pro infixní tvar formule. Vložení znaku negace proběhne vždy, pokud se při procházení stromu narazí na negovaný uzel (atribut *negation* nabývá hodnoty true). V případě, že se

nejedná o výrokovou proměnnou, je nutné ještě vložit závorky. Kulaté závorky se také vkládají, pokud má operátor levého/pravého podvýrazu nižší prioritu než je priorita aktuálně čteného uzlu.

3.4.4 Převod do disjunktivní/konjunktivní normální formy

V rámci ekvivalentních úprav lze vloženou formuli převádět do disjunktivní normální formy (DNF) či konjunktivní normální formy (KNF). Implicitně je v aplikaci nastaven převod do DNF. Takto upravená formule má tvar disjunkce elementárních konjunkcí (viz. kapitola 2.5). Pomocí DNF lze zjistit, zda je formule tautologie či kontradikce. KNF je duální k DNF. Tato forma se skládá z konjunkce elementárních disjunkcí. Formuli převádíme do KNF, chceme-li provádět dokazování při použití rezoluční metody, ale jak si popíšeme později (viz. kapitola 3.5.1), není potřeba při tomto důkazu použít algoritmus pro převod syntaktického stromu do KNF. Vraťme se nyní ale zpět k ekvivalentním úpravám. Pro přepis formule do DNF nebo KNF jsou v aplikaci implementovány dvě pravidla a tím je distributivní zákon pro konjunkci a disjunkci:

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$
$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

Oba tyto distributivní zákony řeší metoda *distribution* třídy **DNFKNF**. Kód této metody by se dal popsat jako jeden obecný distributivní zákon, který převádí danou formuli do DNF nebo KNF podle toho, pro jaký převod se uživatel rozhodl. V konstruktoru třídy DNFKNF se totiž předávají dva znaky jako parametry a ty určují, který znak bude hlavní (spojuje jednotlivé podformule) a který vedlejší (vyskytuje se v jednotlivých podformulích). Pro převod syntaktického stromu do normální formy je použit průchod typu Pre-order reprezentovaný metodou *preOrder*. Nyní si ukážeme na následujícím příkladě postup při provádění převodu syntaktického stromu do DNF.

Příklad 3.3
Mějme formuli p& (q | r).

Program nejprve zavolá metodu *solveDNFKNF* třídy DNFKNF, která je hlavní spouštěcí metodou při převodu do normální formy. V této metodě se volá metoda *preOrder*, které je předán v parametru celý syntaktický strom. Před samotnou aplikací distributivního zákona se volá metoda *modifyNode*, která modifikuje syntaktický strom (pokud je to nutné) takovým způsobem, že do levého podstromu přiřadí pouze jednu formuli, která obsahuje alespoň jednu logickou spojku (přednost mají podformule, které se vyskytují nejvíce vlevo). Všechny ostatní podformule přesune do pravého podstromu. Je-li už taková podformule v levém podstromu obsažena, není třeba provádět modifikaci. Důvod je ten, že se v další části rekurzivní metody *preOrder* analyzuje pouze levý podstrom syntaktického stromu a podle něj se aplikuje distributivní zákon. Zadaná formule se tedy převede na tvar (q | r)& p, kterou již lze upravit v aplikaci.

Samotnou distribuci zastupuje metoda *distribution*. Tato metoda má dva parametry – *leftNode* a *completNode*. *LeftNode* označuje levý podstrom výrazu určeného k distribuci, *completNode* je tento výraz. Nejprve se uloží do pomocného uzlu pravý podstrom

výrazu, který se později spojí s každým potomkem levého podstromu. Pokud jsou tito potomci přímo atomy, distribuce je ukončena. V opačném případě se distribuce provádí dále. Změnu logické spojky zajišťuje metoda *changeOperator*. Výsledná formule odpovídá pravidlu převodu do DNF a vypadá následovně:

$$(p\& q) \mid (p\& r)$$

Syntaktický strom takto převedený do DNF vypadá tak, že od kořene směrem dolů jsou pouze disjunkce a platí, že od první konjunkce se už vyskytují pouze konjunkce nebo výrokové symboly (s negací či bez negace). U převodu syntaktického stromu do KNF je tomu přesně opačně – od kořene směrem dolů jsou pouze konjunkce a platí, že od první disjunkce se už vyskytují pouze disjunkce nebo výrokové symboly. ■

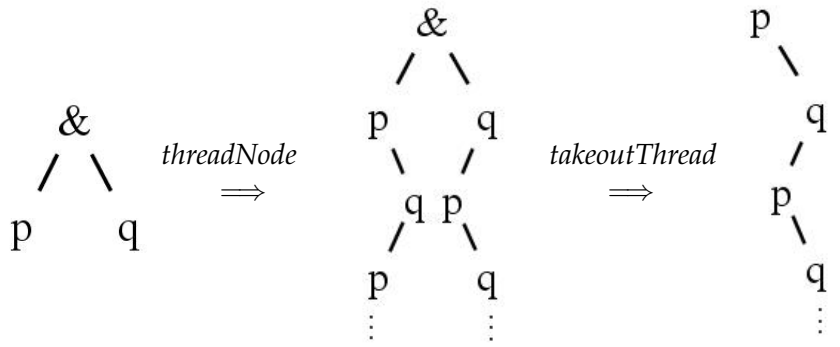
3.4.5 Minimalizace formule

Máme-li formuli v DNF nebo KNF, lze ji určitými pravidly minimalizovat, což znamená zjednodušení formule. Zjednodušujeme takovým způsobem, že dosazujeme hodnoty (0 nebo 1) za výrokové proměnné. Tímto postupem lze dojít k závěru, zda je formule tautologie nebo kontradikce. Mezi zákony, které minimalizují formuli, patří: **absorpce**, **absorpce negace**, **rozšíření**, **komplementarita** a **idempotence**.

absorpce	absorpce negace	rozšíření	komplement.	idempotence
$p\wedge(p\vee q) \equiv p$	$p\wedge(\neg p\vee q) \equiv p\wedge q$	$p \equiv p\vee(q\wedge\neg q)$	$p\wedge\neg p \equiv 0$	$(p\wedge p) \equiv p$
$p\vee(p\wedge q) \equiv p$	$\neg p\wedge(p\vee q) \equiv \neg p\wedge q$	$p \equiv p\wedge(q\vee\neg q)$	$p\vee\neg p \equiv 1$	$(p\vee p) \equiv p$
	$p\vee(\neg p\wedge q) \equiv p\vee q$			
	$\neg p\vee(p\wedge q) \equiv \neg p\vee q$			

Tabulka 3: Pravidla pro minimalizaci formule

Minimalizaci formule provádějí v aplikaci dvě třídy – **Minimize** a **SortReduce**. Tyto třídy implementují výše zmíněné pravidla pro utvoření minimální formy formule. Třída Minimize provádí absorpci, absorpci negace a rozšíření. Metoda solveReduce třídy SortReduce provádí komplementaritu a idempotenci. Obě tyto třídy jsou použity při provádění ekvivalentních úprav, avšak lze jejich funkcionalitu zakázat v menu aplikace a tak zamezit provádění minimalizace dané formule. Třída Minimize definuje metodu *solveMinimize*. Tato metoda zaobaluje celou funkčnost třídy. Nejdříve se převedou všechny disjunkty či konjunkty syntaktického stromu na jakási vlákna, která jsou vytvořena metodou *threadNode*. V těchto vláknech se ještě vyskytují jejich logické spojky (disjunkce nebo konjunkce). Další metoda, která zpracovává vlákno, se nazývá *takeoutThread*. Oproti *threadNode* vyjme tato metoda z vlákna levou část (eliminace logických spojek) a v dalších krocích lze provést absorpci dle implementovaných pravidel metody *absorbe*. Po ukončení minimalizace jsou vlákna převedena do původních struktur uzlů syntaktického stromu. Pro představu si graficky uvedeme postupný převod formule p&q na vlákno.



Obrázek 8: Původní syntaktický strom

Obrázek 9: Vlákno 1 (s log. spojkami)

Obrázek 10: Vlákno 2 (bez log. spojek)

Třída **SortReduce** také implementuje ještě jednu metodu, která třídí znaky v syntaktickém stromě podle jejich velikosti (abecedně od prvního po poslední). Metoda je pojmenována *solveSort*. Tento algoritmus je důležitý při porovnávání stromů mezi sebou.

3.4.6 Úložiště výsledků řešení

Pro ukládání výsledků jsou k dispozici dvě třídy - **ResultEkviv** a **ResultRezol**. Obě třídy implementují pouze statické proměnné (a metody). Tyto proměnné jsou statické proto, aby bylo možné s jejich obsahem pracovat i dále v aplikaci a následně zobrazovat jejich obsah v uživatelském rozhraní. Třída ResultRezol slouží k ukládání výsledků příkladů řešených pomocí rezoluční metody. Tuto třídu si popíšeme v následující kapitole (viz. kapitola 3.5.6).

Třída ResultEkviv je určena jako úložiště pro výsledky důkazů ekvivalentními úpravami. Proměnná *stepsList* typu seznam zachycuje syntaktický strom procházející jednotlivými transformacemi ekvivalentními úpravami. Jinými slovy, každý záznam tohoto seznamu je reprezentací vstupní formule převedené na syntaktický strom a pozměněné určitými ekvivalentními úpravami. Proměnné *steps* a *result* obsahují textovou reprezentaci řešení dané formule. V poli řetězců *steps* jsou uloženy jednotlivé kroky při postupu k řešení, *result* pak obsahuje samotné řešení dané formule.

3.4.7 Výstup aplikace

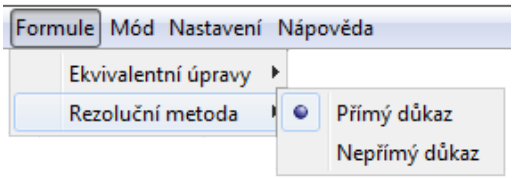
Tato podkapitola zachycuje jednotlivé kroky řešení při provádění důkazu ekvivalentními úpravami v aplikaci. Tyto kroky jsou zobrazeny ve výstupním textovém poli. Výstup lze z aplikace uložit do textového souboru.

Mějme formuli p = q. Tato formule je převedena do DNF a jsou na ni aplikovány zákony pro minimalizaci formule. Výsledné řešení vypadá následovně:

1. (p>q)&(q>p)	Eliminace ekvivalence [A=B = (A>B)&(B>A)]
2. (∼p	q)&(q>p)	Eliminace implikace [A>B = ∼A	B]
3. (∼p	q)&(∼q	p)	Eliminace implikace [A>B = ∼A	B]
4. (∼p&(∼q	p))	(q&(∼q	p))	Distributivní zákon [A&(B	C) = (A&B)	(A&C)]
5. (∼p& q)	(p&∼p)	(q&(∼q	p))	Distributivní zákon [A&(B	C) = (A&B)	(A&C)]
6. (∼p& q)	(0&∼p)	(q&(∼q	p))	Komplementarita [A&∼A = 0]
7. (∼p& q)	0	(∼q&q)	(p&q)	Distributivní zákon [A&(B | C) = (A&B)	(A&C)]
8. (∼p& q)	0	(0&q)	(p&q)	Komplementarita [A&∼A = 0]
Výsledek: (∼p&∼q)	(p&q)		

3.5 Důkazy pomocí rezoluční metody

Doposud byly podrobně popsány transformace formule, které jsou nezbytné při provádění důkazů ekvivalentními úpravami. Nyní se zaměřme na důkazovou metodu, která ověřuje platnost úsudku. Tuto metodu označujeme jako rezoluční metoda. Hlavním kritériem pro použití rezoluční metody je fakt, že řešený logický výraz musí být v KNF. A proto je nutné se postarat o převod logického výrazu do KNF ještě před tím, než začneme cokoliv dokazovat. Chceme-li provádět důkaz pomocí rezoluční metody, nabízí se možnost dvou odlišných postupů, jak tento důkaz provést. Prvním z nich je přímý důkaz, ten druhý je nepřímý důkaz. Oba tyto postupy jsou implementovány do systému a aplikace umožňuje uživateli v menu přepínání mezi nimi. Výsledkem důkazu rezoluční metody je zjištění, zdali je zadaná množina klauzulí splnitelná či nikoli.



Obrázek 11: Menu - Rezoluční metoda

3.5.1 Sestavení syntaktického stromu

Aby mohl být vytvořen syntaktický strom určený ke zpracování rezoluční metodou, musí být nejdříve sestavena formule, která bude reprezentovat daný logický výraz a bude vzorem pro konstrukci tohoto stromu. Sestavení formule probíhá v aplikaci tak, že se výrazy jednotlivých klauzulí na vstupu poskládají do jednoho řetězce, který je následně předán v parametru ke zpracování rezoluční metodou. K sestavení řetězce je použita metoda *createRezFormula*. Tato metoda je volána v cyklu (počet opakování je dán počtem klauzulí) a za každý logický výraz dané klauzule je přidán znak &, jelikož jsou všechny klauzule navzájem k sobě v konjunktivním vztahu. Toto je první krok pro převedení formule do KNF. Nicméně výrazy jednotlivých klauzulí nemusí obsahovat pouze spojky disjunkce, takže tvar KNF ještě nemusí být splněn.

Formule je následně předána syntaktickému analyzátoru, který z ní vytvoří syntaktický strom. Strom je upraven metodami *solveConst* (vyhodnocení logických konstant) a *solveIDC* (odstranění negace směrem na proměnné, eliminace logických spojek). Takto upravený strom již splňuje tvar KNF a lze s ním provést důkaz rezoluční metodou.

3.5.2 Postup provádění rezoluční metody

Provádění důkazů pomocí rezoluční metody zajišťuje třída **Rezolution**. Tato třída implementuje postupy provádění přímého důkazu prostřednictvím funkce *PD* a nepřímého důkazu prostřednictvím funkce *ND*. Jako návratová hodnota obou funkcí je volání funkce *solveRezol*, které je předán syntaktický strom, takto funkce provede potřebné operace a vrátí strom upravený rezoluční metodou. Funkce *solveRezol* je stěžejní funkcí celé třídy Rezolution. Tato funkce řeší transformaci syntaktického stromu, řeší rezoluci mezi jednotlivými klauzulemi a všechny získané výsledky ukládá do statických proměnných třídy **ResultRezol**.

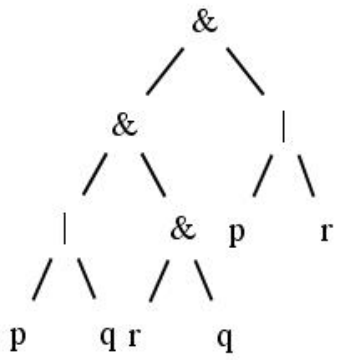
Ještě před samotným dokazováním musí být ale syntaktický strom (jeho struktura) převeden do tvaru, přijatelného ke zpracování jednotlivými metodami třídy. Následně již lze provádět se stromem potřebné operace.

3.5.3 Transformace struktury syntaktického stromu

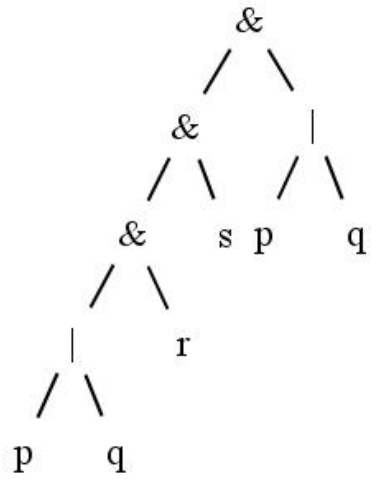
Syntaktický strom předaný třídě **Rezolution** již splňuje tvar KNF, nicméně je potřeba jeho strukturu upravit do podoby, se kterou se bude lépe pracovat v dalších krocích při provádění důkazů. Tuto transformaci zajišťuje metoda *transform*. Výsledkem transformace je pak strom, jehož pravé uzly obsahují pouze disjunkce nebo výrokové symboly. V levých uzlech stromu se vyskytují navíc i logické spojky konjunkce. Touto úpravou stromu lze následně porovnávat jednotlivé vstupní klauzule mezi sebou. Následující příklad graficky popisuje syntaktický strom před a po provedení transformace metodou *transform*.

Příklad 3.4
Mějme množinu klauzulí $M = \{p \mid q, r \,\&\, q, p \mid r\}$.

Předpokládejme, že již v aplikaci proběhla SA, vytvořený syntaktický strom byl převedený do KNF a nyní je předán metodě *transform*.



transform $\Longrightarrow$



Obrázek 12: Původní strom

Obrázek 13: Transformovaný strom

■

3.5.4 Principy metody *PD*

V případě přímého důkazu se ze syntaktického stromu vyjme pravý podstrom, který představuje dokazovanou formuli (závěr) vstupní množiny klauzulí, a uloží se do pomocného uzlu *conclusion*, který se porovnává v další části programu se vzniklými rezolventy.

Prvním krokem, při provádění samotného důkazu, je ověření, zdali ze vstupní množiny klauzulí neplyne přímo splnitelnost této množiny. To znamená, že se porovnávají vstupní klauzule se závěrem. Toto porovnávání implementuje metoda *compareClauses*. V kladném případě jsou výsledky uloženy a rezoluční metoda je úspěšně ukončena. V opačném případě provede metoda *solvePremise* porovnání všech vstupních klauzulí mezi sebou navzájem. Vytváří rezolventy z dvojic klauzulí, které mají alespoň jeden shodný literál s navzájem opačným znaménkem negace a tyto rezolventy ukládá do bufferu (*bufferNode*). Ten slouží jako úložiště všech takto vytvořených rezolvent. Po ukončení metody *solvePremise* jsou v bufferu uloženy všechny rezolventy, které vznikly porovnáváním jednotlivých klauzulí.

Následující krok je závislý na obsahu zmiňovaného bufferu. Pokud není prázdný (a doposud nebyla dokázána platnost množiny), je volána v cyklu while další metoda, která je pojmenována *solveRezolvent*. Tato metoda vybírá z bufferu jednotlivé rezolventy a porovnává je se závěrem. Nastat mohou opět dvě situace. První, kdy bude rezolventa totožná se závěrem, což vede k ukončení rezoluční metody nebo druhá, kdy se rezolventa nebude se závěrem shodovat. V druhé z těchto situací se opět volá metoda *compareClauses*, ovšem nyní se porovnávají doposud uložené klauzule (celý syntaktický strom) s vybranou rezolventou a případně se vytváří další rezolventy, které se ukládají do bufferu. V obou ze zmíněných situací se rezolventa automaticky připojuje k syntaktickému

stromu. Volání metody *solveRezolvent* je ukončeno v momentě, kdy je buffer určený k ukládání rezolvent prázdný nebo byla dokázána splnitelnost množiny klauzulí.

3.5.5 Principy metody *ND*

Při nepřímém důkazu funkce také upravuje syntaktický strom způsobem, že z něj vyjme pravý podstrom, který uloží do uzlu *conclusion*, navíc ale tento uzel zneguje, odstraní z něj negace směrem na proměnné, eliminuje logické spojky a poté jej zpět uloží do syntaktického stromu metodou *addNodeToRoot*. Nakonec se uzel *conclusion* vymaže (přiřazení hodnoty null).

Následující kroky rezoluční metody jsou velmi podobné přímému důkazu. Také se nejdříve porovnávají vstupní klauzule mezi sebou metodou *solveRezol* a všechny nově vzniklé rezolventy jsou uloženy do bufferu (*bufferNode*). Rozdíl je ale v porovnávání jednotlivých klauzulí mezi sebou. Zatímco u přímého důkazu se porovnávají klauzule se závěrem, u nepřímého důkazu se porovnávají klauzule mezi sebou a hledají se takové, které jsou navzájem ve sporu (stejný literál ale opačná spojka negace). Pokud program na takové dvě klauzule narazí, důkaz rezoluční metodou je ukončen a všechny výsledky jsou uloženy k následnému zobrazení řešení.

Při výběru jednotlivých rezolvent z bufferu se porovnávají doposud uložené klauzule (celý syntaktický strom) s vybranou rezolventou. Vznikne-li při porovnávání nová rezolventa, je uložena do bufferu sloužící pro dočasné uložení rezolvent, které vzniknou rezolucí nově vytvořených rezolvent. Tento buffer je v kódu označen jako *bufferNodeND*. Poslední dvě písmena v názvu metody značí, že se tento buffer používá pouze pro metodu nepřímého důkazu rezoluční metodou. Při vytváření takto vzniklých rezolvent je rekurzivně volána metoda *compareClauses*, která porovnává vždy nově vytvořenou rezolventu s celým syntaktickým stromem. Tak je zajištěno co nejefektivnější porovnávání klauzulí a rezolvent mezi sebou.

3.5.6 Úložiště výsledku řešení

K ukládání výsledků, které vzejdou z důkazů pomocí rezoluční metody, slouží třída **ResultRezol**, která implementuje pouze statické proměnné (a metody).

Proměnná *nodeList* je seznamem všech klauzulí, které byly vloženy a následně i vytvořeny aplikací při rezoluční metodě. Jednotlivé klauzule mají strukturu syntaktického stromu. Tato proměnná typu seznam slouží především ke kontrole výsledků, které vkládá uživatel v interaktivním režimu aplikace. Proměnná *nodeListString* je opět seznamem všech klauzulí vytvořených při rezoluční metodě, ovšem jednotlivé záznamy jsou typu **String**. Tento seznam je využit jako efektivní volba při ukládání řešeného příkladu do formátu XML a TXT. Poslední proměnná typu seznam nese název *rowsNum*, která obsahuje všechna čísla řádků, se kterými byla provedena rezoluce a důsledkem toho vznikla rezolventa, která se nachází v řešení. Ostatní proměnné slouží spíše jako pomocné prvky při dynamické tvorbě jednotlivých klauzulí v grafickém prostředí aplikace. Jsou to proměnné *counterFormula* (počet klauzulí vstupní množiny) a *counterConclusion* (počet klauzulí dokazované formule). Poslední dvě proměnné typu boolean určují, zdali řešená

množina formulí je splnitelná (proměnná *ok*) a zdali v této množině došlo při provádění důkazu ke sporu (proměnná *sharp*).

3.5.7 Výstup aplikace

Následující ukázka zachycuje postupné řešení příkladu při provádění důkazu pomocí rezoluční metody v aplikaci. Postup při řešení příkladu je zobrazen v aplikaci v grafické podobě, nicméně lze celý příklad (i se správným řešením) uložit do textového souboru.

Na vstupu je množina klauzulí {f | c | ∼s, ∼c | p, s} a závěr ∼f>p. Na tyto klauzule je aplikován nepřímý důkaz rezoluční metody. Výsledné řešení vypadá následovně:

f | c | ∼s
∼c | p
s
∼f>p závěr

1.	f | c | ∼s	
2.	∼c | p	
3.	s	
4.	∼f	(negovaný závěr)
5.	∼p	(negovaný závěr)
6.	f | p | s	(1, 2)
7.	f | p	(3, 6)
8.	p	(4, 7)
9.	#	(5, 8)

3.6 Kontrola řešení - interaktivní mód aplikace

Interaktivní mód aplikace nabízí uživateli postupně provádět jednotlivé kroky řešení daného příkladu. Uživatel si tedy může zkontrolovat, zdali postupuje při řešení správnou cestou. Aplikace vždy vyhodnotí vloženou formuli a oznámí její správnost či nesprávnost. Aplikace je také schopna nabídnout nápovědu, která má za úkol navést uživatele na správný postup řešení jednotlivých kroků. V případě, že by ani nápověda nepomohla, lze kdykoliv zobrazit aktuální krok a postupovat dále v řešení příkladu. Kontrola vstupní formule a následné ověření, zdali se jedná o správné řešení dílčího kroku, probíhá přímo ve formulářích, ve kterých je implementováno uživatelské rozhraní. Konkrétně se jedná o formuláře **EkvIA** (kontrola ekvivalentních úprav) a **MainFrame** (kontrola rezoluční metody). Algoritmus, který implementuje tuto kontrolu je spjat s akčními metodami komponent.

3.6.1 Kontrola řešení - ekvivalentní úpravy

Ve formuláři **EkvIA** je definována akční metoda *jB_ekvCheckActionPerformed* komponenty *jB_check*. V této metodě probíhá porovnávání formule na vstupu s výsledným ře-

šením, které je uloženo v seznamu *stepsList* třídy **ResultEkviv**. Záznamy tohoto seznamu představují jednotlivé kroky transformace formule, která je převedena do syntaktického stromu. Jelikož je tento seznam statický, není potřeba vytvářet instanci dané třídy. Pokud tedy uživatel zadá na vstup formuli určenou ke kontrole, nejdříve musí být převedena na syntaktický strom, ze stromu musí být odstraněny konstanty a následně je na tento strom aplikována metoda, která seřadí jednotlivé výrokové proměnné vzestupně podle abecedy. Po všech těchto transformacích lze zahájit porovnávání. Tento proces zajišťuje funkce *compareNodes* třídy **ControlTree**. Funkce vrací hodnotu true, pokud se porovnávané syntaktické stromy sobě rovnají, v opačném případě tato funkce vrací false. V případě, že jsou si formule navzájem ekvivalentní, odstraní se aktuální záznam ze seznamu *stepsList* (již jej dále v programu nebude potřeba) a v uživatelském rozhraní se do výstupního textového pole vloží správně vyřešený krok. Následně lze provádět kontrolu dalších kroků v aplikaci. V případě, že si porovnávané formule navzájem ekvivalentní nejsou, proběhne porovnání se všemi záznamy seznamu *stepsList*. Uživatel přeci nemusí postupovat striktně podle vyřešených kroků v daném pořadí, tak jak jsou uloženy v systému, ale smí některé kroky při řešení přeskočit. Je-li dané řešení kroku správné, algoritmus odstraní všechny předcházející kroky, které jsou uloženy v seznamu *stepsList*. Tímto způsobem je zajištěna univerzálnost postupu při řešení, kdy řešitel není přímo omezen konkrétním postupem, který vede ke správnému výsledku.

3.6.2 Kontrola řešení - rezoluční metoda

Ve formuláři **MainRezol** je definována akční metoda *jB_rezIACheckActionPerformed* komponenty *jB_ekvCheck*. Tato metoda porovnává formuli na vstupu s výsledným řešením, které je uloženo ve statickém seznamu *stepsList* třídy **ResultRezol**. Tento seznam obsahuje všechny klauzule rezoluční metody. Tyto klauzule jsou převedeny na strukturu syntaktického stromu a uloženy v seznamu jako jednotlivé záznamy. Porovnávání je velmi podobné algoritmu pro kontrolu ekvivalentních úprav. Formule na vstupu musí být nejdříve převedena na syntaktický strom, poté proběhne eliminace logických spojek, odstranění konstant ze stromu a nakonec je strom tříděn metodami *solveSort* a *solveReduce*. Takto upravený strom se porovnává s řešením pomocí metody *compareNodes*. Pokud jsou řešení shodná, aplikace oznámí správnost řešení. V opačném případě je uživatel upozorněn na špatně řešený krok.

3.7 Návrh implementace

Vývoj aplikace Logik byl poměrně dlouhodobou záležitostí. Nejdříve jsem se musel rozhodnout, na jaké platformě by měla být aplikace vytvořena. Díky zkušeností z předešlých let s programovacím jazykem JAVA jsem se rozhodl právě pro tento jazyk. Implementace probíhala v prostředí Netbeans IDE 7.0.1.

Při implementaci systému jsem využil znalostí z předmětu *Úvod do teoretické informatiky*, který jsem absolvoval v bakalářském studiu. Navíc bylo potřeba nastudovat základy tvorby překladačů [3, 4], čehož jsem hlavně využil při konstrukci syntaktického analyzátoru. Inspirací při tvorbě algoritmu, který provádí důkazy ekvivalentními úpra-

vami, mi byla aplikace *Bachelor* [5], která implementuje (v jazyce Pascal) metody, které provádí transformace se vstupní formulí.

Pro tvorbu grafického rozhraní jsem využil knihovnu *Swing*. Tato knihovna slouží k vytváření uživatelských prvků na platformě JAVA. Aplikace je konstruována tak, aby se grafické prostředí přizpůsobilo aktuálnímu nastavení operačního systému, takže vzhled celého grafického rozhraní je dán systémovým nastavením prvků operačního systému.

Zdrojový kód aplikace je rozdělen do tříd a tyto třídy jsou řazeny do jednotlivých balíků podle toho, jakou funkčnost v aplikaci zastupují.

Hlavní balíky aplikace:

- **Control** – obsahuje třídu s pomocnými metodami, které jsou využívány v celé aplikaci
- **Database** – obsahuje třídy pro ukládání příkladů do XML a TXT souborů
- **GUI** – implementuje grafické rozhraní aplikace (jednotlivé formuláře a jejich komponenty)
- **GUI.Image** – shromažďuje všechny obrázky, které jsou využity v grafickém rozhraní aplikace
- **Interface** – obsahuje rozhraní, která definují symboliku důležitých znaků a jejich prioritu, pravidla při ekvivalentních úpravách a také chyby, které mohou nastat v aplikaci
- **Main** – obsahuje třídu se spouštěcí metodou *main* a také kořenovou třídu **Root**, která zahajuje výpočet všech operací v aplikaci
- **Model** – obsahuje třídy reprezentující objekty, se kterými se pracuje při výpočtech v aplikaci
- **Operations** – definuje třídy, které provádějí transformace se syntaktickým stromem
- **Result** – slouží jako úložiště pro výsledky, obsahuje dvě třídy, které uchovávají postupy a výsledky řešení ve statických proměnných
- **Thread** - obsahuje třídy, které reprezentují vlákna, ve kterých se spouštějí výpočty, vlákna jsou implementována z důvodu možnosti přerušení provádění výpočtu

3.8 Ovládání aplikace

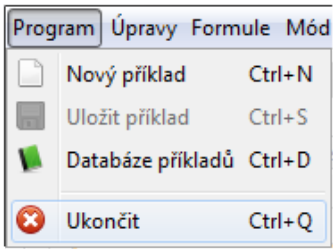
Aplikace Logik je navržena jako jedno-okenní aplikace, takže všechny operace k provedení výpočtu příkladů a zobrazení jejich výsledků se provádějí v hlavním okně, které nelze duplikovat. Výjimkou je interaktivní okno pro řešení formule ekvivalentními úpravami, kdy aplikace vytváří ještě jedno přídavné okno, ve kterém se provádí konkrétní výpočet. Hlavní okno se skládá ze tří oddělených částí: **menu**, **nástrojová lišta** a **hlavní panel pro výpočet**.

3.8.1 Menu

Menu aplikace je vytvořeno tak, aby se v něm mohl uživatel co nejjednodušeji a nejrychleji orientovat. Obsahuje tyto položky: **Program**, **Úpravy**, **Formule**, **Mód**, **Nastavení**, **O programu**.

3.8.1.1 Program

Položka Program obsahuje volby, které poskytují tvorbu příkladů v aplikaci. Jedná se o volby: **Nový příklad**, **Uložit příklad**, **Databáze příkladů**, **Ukončit**.

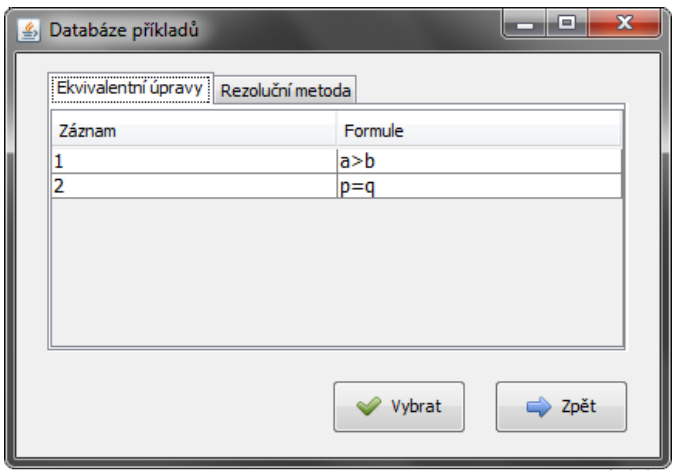


Obrázek 14: Menu - Program

- **Nový příklad**
  Zahájení nového výpočtu čili příprava prostředí pro výpočet nového příkladu. Volbou této položky se vymažou všechna textová pole (karta ekvivalentní úpravy), případně se vyprázdní panel pro editaci klauzulí (karta rezoluční metoda).

- **Uložit příklad**
  Tato volba ukládá aktuálně vyřešený příklad do textového souboru. Uživateli je nabídnut dialog, pomocí kterého vytvoří název souboru a zadá cestu k jeho uložení.

- **Databáze příkladů**
  Aplikace si všechny správně vyřešené příklady ukládá do své databáze, kterou představuje soubor formátu XML. Tento soubor je automaticky vytvořen při prvotním vložením správně zadaného příkladu. Jelikož mohou být příklady dvou typů (ekvivalentní úpravy, rezoluční metoda), musí aplikace vytvořit i dva různé XML soubory. Jejich názvy jsou *fileEkviv* a *fileRezol*. Tyto soubory jsou uloženy v adresáři *LogikDatabase*, který generuje aplikace automaticky.
  Všechny příklady, které jsou uložené v těchto souborech, lze zpětně načít do systému. K tomuto účelu slouží položka Databáze příkladů, pomocí které lze zobrazit všechny doposud správně vyřešené příklady. Jednoduchým výběrem a potvrzením tlačítkem **Vybrat** se příklad zpětně načte do systému i se svým řešením.



Obrázek 15: Menu - Databáze příkladů

- **Ukončit**
  Tato položka ukončuje práci s aplikací Logik.

3.8.1.2 Úpravy

Nabídka **Úpravy** poskytuje klasické volby pro úpravu textu. Jsou jimi volby **Zpět** a **Vpřed**.

3.8.1.3 Formule

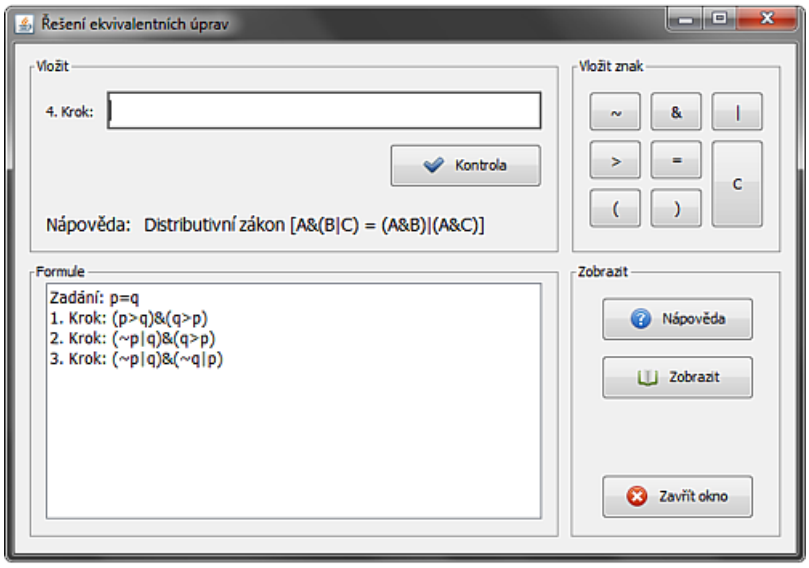
Pomocí této položky lze ovlivnit nastavení pro jednotlivé oblasti důkazů. Jejími volbami jsou: **Ekvivalentní úpravy** a **Rezoluční metoda**.

3.8.1.4 Mód

Mód aplikace vyjadřuje, jakým způsobem se budou zobrazovat výsledky vložených příkladů. V menu lze vybírat mezi základním a interaktivním módem. Rozdíl mezi těmito módy je v míře interaktivity, kterou vytváří aplikace mezi uživatelem.

- **Základní mód**
  V tomto módu lze řešit příklady VL, zobrazovat jejich výsledky a postup při řešení. Uživatel si tak jednoduše může ověřit správnost řešení daného příkladu.

- **Interaktivní mód**
  V tomto módu je uživatel přímo zapojen do řešení příkladů a aplikace kontroluje jeho postupy a závěry. Uživatel si tak může prověřit své poznatky z oblasti VL v praxi. Během řešení příkladů nabízí aplikace možnost nápovědy, která zobrazuje jaký zákon aplikovat (ekvivalentní úpravy) či jaké řádky spojit (rezoluční metoda). V případě ekvivalentních úprav je v tomto režimu uživateli nabídnuto nové okno, ve kterém se provádí řešení daného příkladu. Toto okno je zobrazeno na obrázku 16.



Obrázek 16: Interaktivní mód

3.8.1.5 Nastavení

V této položce se nachází volba **Písmo**, pomocí níž lze měnit velikost a styl písma ve vstupních a výstupních textových polích v aplikaci. Dialog pro změnu písma byl převzat jako volně šiřitelný kód. Upraveny byly pouze popisky jednotlivých komponent dialogu (z anglického jazyka na český). Celý soubor s příponou *.java* byl získán z tohoto **odkazu**.

3.8.1.6 Nápověda

Nápověda obsahuje položku **O aplikaci**, která zobrazuje dialog, ve kterém se nacházejí informace o vytvořené aplikaci.

3.8.2 Nástrojová lišta

Tato lišta má za úkol nabízet nejvíce používané položky menu nabídky. Jsou jimi: **Nový příklad**, **Uložit příklad**, **Databáze příkladů**, **Zpět**, **Vpřed**. Dále je zde také tlačítko, která se vztahuje k samotnému výpočtu příkladů. Toto tlačítko je označeno jako **Stop** a jeho funkcí je přerušení probíhajícího výpočtu.

Obrázek 17: Nástrojová lišta

3.8.3 Hlavní panel

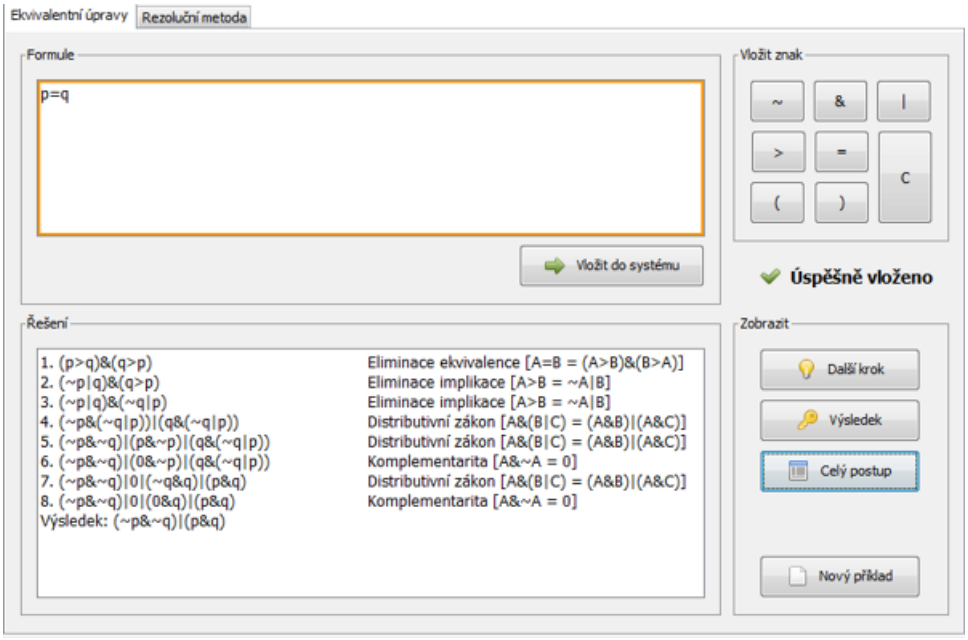
Tento panel je nejdůležitější částí grafického rozhraní aplikace. Zde se provádí výpočet vstupních formulí a pomocí něj lze zobrazit výsledky ve výstupních komponentech aplikace. Panel se skládá ze dvou karet, mezi kterými lze přepínat. Karta **Ekvivalentní úpravy** slouží k provedení důkazů ekvivalentními úpravami a karta **Rezoluční metoda** slouží k provedení důkazů rezoluční metodou.

3.8.3.1 Ekvivalentní úpravy

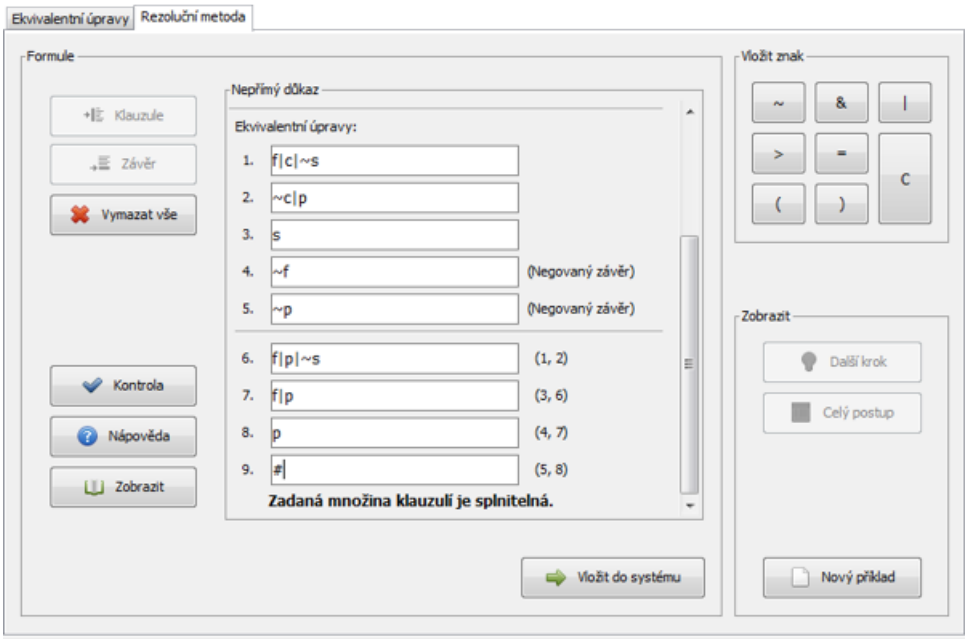
Tento panel se skládá z dvojice textových polí, které slouží jako vstupní a výstupní pole. V poli **Formule** se vytváří vstupní formule určená ke zpracování systémem. V poli **Řešení** se pak zobrazují jednotlivé kroky řešení a výsledná formule po provedení náležitých ekvivalentních úprav. Pro vložení formule do systému je vytvořeno tlačítko **Vložit** do systému, které spouští celý proces úprav formule. V pravém horním rohu se nachází skupina tlačítek, která má za úkol zjednodušit vytváření vstupní formule. Pomocí tlačítka označeného velkým písmenem **C** se provádí přemazání vstupního pole Formule. Poslední skupina tlačítek je umístěna v pravém dolním rohu. Pomocí těchto tlačítek lze zobrazit výsledné řešení, jednotlivé kroky řešení nebo rovnou celý postup řešení. Snímek programu, který reprezentuje hlavní panel pro výpočet pomocí ekvivalentních úprav, je na obrázku 18.

3.8.3.2 Rezoluční metoda

Hlavním prvkem tohoto panelu je grafické pole, které je označeno názvem Přímý důkaz nebo Nepřímý důkaz (podle volby postupu rezoluční metody v menu aplikace). V tomto poli se graficky vytvářejí textové pole pro zadávání formulí jednotlivých klauzulí. Vytváření těchto textových polí režíruje skupina tlačítek v levém horním rohu panelu. Jedná se o trojici tlačítek, pomocí nichž lze vytvářet vstupní klauzule (**Klauzule**), dokazovanou formuli (**Závěr**) a třetím tlačítkem se provádí vymazání všech vytvořených klauzulí (**Vymazat vše**) v grafickém poli. Níže pod těmito tlačítky je vytvořena další skupina tlačítek, která obsluhuje aplikaci při výpočtu množiny formulí v interaktivním módu. Jedná se o tlačítka **Kontrola** (provádí kontrolu uživatelem vložené formule), **Nápověda** (zobrazuje nápovědu k řešení aktuálního kroku důkazu) a **Zobrazit** (zobrazuje aktuální krok řešení). Další skupiny tlačítek jsou totožné jako u ekvivalentních úprav. Nachází se zde tlačítka, která slouží k rychlému vytváření formule a také tlačítka, pomocí nichž lze zobrazit výsledky a to buď jednotlivě, nebo vše najednou. Nesmíme také opomenout tlačítko, kterým se vkládá vstupní množina formulí do systému. Toto tlačítko se nachází ve spodní části panelu Rezoluční metoda. Snímek programu, reprezentující hlavní panel pro výpočet pomocí rezoluční metody, je na obrázku 19.



Obrázek 18: Hlavní panel - Ekvivalentní úpravy



Obrázek 19: Hlavní panel - Rezoluční metoda

4 Závěr

Výroková logika je jednou z hlavních částí matematické logiky. Tato práce se zabývá danými oblastmi VL po teoretické i praktické stránce. Teoretická část je popsána v úvodních kapitolách. Hlavním cílem bylo vytvořit algoritmus, který řeší vybrané příklady z VL pomocí důkazových metod: ekvivalentní úpravy a rezoluční metoda. Pro zavedení tohoto algoritmu do praxe byl implementován interaktivní systém, který vizuálně zpracovává jednotlivé příklady a nabízí uživateli výsledné řešení. Tento systém je navržen tak, aby seznámil uživatele s problematikou daného příkladu a navrhl mu možné řešení v případě potřeby. Při návrhu a implementaci systému jsem vycházel a nechal se inspirovat již vytvořenou aplikací *Bachelor* [5], která provádí řešení příkladů VL pomocí ekvivalentních úprav. Tato aplikace mi byla také velmi dobrou oporou při sestavování syntaktického analyzátoru, který je nezbytný pro SA vstupního řetězce. Systém implementovaný mnou ovšem převyšuje aplikaci *Bachelor* v řešení vybraných příkladů tím, že nabízí interaktivní mód, ve kterém lze provádět kontrolu jednotlivých kroků řešení a také zobrazovat k těmto krokům vlastní nápovědu. Navíc je systém doplněn o algoritmus, který provádí důkazy pomocí rezoluční metody. Samotná myšlenka implementace programu byla iniciována také za účelem vytvoření sbírky řešených příkladů VL, a proto jsou všechny správně vyřešené příklady uloženy do databáze systému, ze které lze příklady zpětně načítat a zobrazit jednotlivé kroky při řešení. Tato práce se může stát podkladem při studiu teoretické informatiky, která se vyskytuje v mnoha programech studia informačních technologií vysokých škol. Písemná část práce také popisuje celkový postup při implementaci systému. Tyto kapitoly mají za úkol přiblížit čtenáři vnitřní stavbu systému za účelem možného rozšíření systému o další skupiny úloh. I když byla tato práce náročná, získal jsem spoustu cenných zkušeností a byla pro mne přínosem.

Ladislav Barabáš

5 Reference

[1] DUŽÍ, Marie. *Matematická logika* [online]. Ostrava: Katedra Informatiky FEI, 2003 [cit. 2012-04-16]. Dostupné z: <http://www.cs.vsb.cz/duzi/Matlogika_Vyber.pdf>

[2] BĚLOHLÁVEK, Radim; VYCHODIL, Vilém. *Diskrétní matematika pro informatiky I* [online]. Olomouc: Univerzita Palackého, Přírodovědecká fakulta, Katedra informatiky, 2006 [cit. 2012-04-16]. Dostupné z: <http://www.kubaz.cz/texty/BelohlavekDMI.pdf>

[3] HABIBALLA, Hashim. *Překladače* [online]. Ostrava: Ostravská univerzita, 2005 [cit. 2012-04-16]. Dostupné z: <http://www1.osu.cz/home/habibal/publ/xprek.pdf>

[4] HABIBALLA, Hashim; VOLNÁ, Eva; FOJTÍK, Rostislav. *Od teorie formálních jazyků k jednoduchému překladači* [online]. Ostrava: Ostravská univerzita, Přírodovědecká fakulta, 2007 [cit. 2012-04-16]. Dostupné z: <http://www1.osu.cz/home/habibal/files/mfi5big.pdf>

[5] HABIBALLA, Hashim. *Bachelor* [online]. Ostrava: Ostravská univerzita, Přírodovědecká fakulta, 1996 [cit. 2012-04-16]. Dostupné z: <http://www1.osu.cz/home/habibal/files/bachelor.zip>

A Obsah CD

Součástí bakalářské práce je CD s tímto obsahem:

- bakalářská práce ve formátu PDF
- projekt implementovaný v prostředí Netbeans IDE 7.0.1
- zdrojové kódy aplikace v jazyce JAVA
- spustitelný soubor, který reprezentuje výsledný program